Efficient Processing of Continuous Join Queries using Distributed Hash Tables

Wenceslao Palma[1], Reza Akbarinia[2], Esther Pacitti[1], Patrick Valduriez[1]

[1]Atlas Team, INRIA and LINA, University of Nantes, France
{FirstName.LastName@univ-nantes.fr, Patrick.Valduriez@inria.fr}
[2]School of Computer Science, University of Waterloo, Canada
{rakbarin@cs.uwaterloo.ca}

Abstract. This paper addresses the problem of computing approximate answers to continuous join queries. We present a new method, called DHTJoin, which combines hash-based placement of tuples in a Distributed Hash Table (DHT) and dissemination of queries using a gossip style protocol. We provide a performance evaluation of DHTJoin which shows that DHTJoin can achieve significant performance gains in terms of network traffic.

1 Introduction

Recent years have witnessed major research interest in data stream management systems. A data stream is a continuous and unbounded sequence of data items. There are many applications that generate streams of data including financial applications [7], network monitoring [23], telecommunication data management [6], sensor networks [5], etc. Processing a query over a data stream involves running the query continuously over the data stream and generating a new answer each time a new data item arrives. Due to the unbounded nature of data streams, it is not possible to store the data entirely in a bounded memory. This makes difficult the processing of queries that need to compare each new arriving data with past ones. We are interested in systems which have limited main memory but that can tolerate an approximate query result which has a maximum subset of the result. An example of such queries is join queries which are very important for many applications. As an example, consider a network monitoring application that needs to issue a join query over traffic traces from various links, in order to monitor the total traffic that is common among three links L1, L2 and L3 over the last 10 minutes. Each link (stream) contains tuples each one with a packet identifier pid and the packet size. This query can be posed using a declarative language such as CQL [3], which is a relational query language for data streams, as follows:

Select sum (L1.size)
From L1[range 10 min], L2[range 10 min], L3[range 10 min]
Where L1.pid=L2.pid and L2.pid=L3.pid

A common solution to the problem of processing join queries over data streams is to execute the query over a sliding window [11] that maintains a restricted number of recent data items. This allows queries to be executed in a finite memory and in an incremental manner by generating new answers when a new data item arrives.

In this paper, we address the problem of computing approximate answers to windowed stream joins over data streams. Our solution involves a scalable distributed sliding window that takes advantage of the free computing power of DHT networks and can be equivalent to thousands of centralized sliding windows. Then, we propose a method, called DHTJoin, which deals with efficient execution of join queries over all data items which are stored in the distributed sliding window. DHTJoin combines hash-based placement of tuples in the DHT and dissemination of queries using a gossip style protocol. We evaluated the performance of DHTJoin through simulation. The results show the effectiveness of our solution compared with previous work.

The rest of this paper is organized as follows. In Section 2, we introduce our system model and define the problem. In Section 3 we describe DHTJoin. Section 4, describes a performance evaluation of our solution through simulation using Simjava. In Section 5, we discuss related work. Finally, Section 6 concludes.

2 System Model and Problem Definition

In this section we introduce a general system model for processing data streams over DHTs, with a DHT model, a stream processing model and a gossip dissemination system. Then, we state the problem.

2.1 DHT Model

In our system, the nodes of the overlay network are organized using a DHT protocol. While there are significant implementation differences between DHTs [19][22], they all map a given key k onto a peer p using a hash function and can lookup p efficiently, usually in $O(log\ n)$ routing hops where n is the number of peers. DHTs typically provide two basic operations [22]: *put(k, data)* stores a key k and its associated *data* in the DHT using some hash function; *get(k)* retrieves the data associated with k in the DHT. Tuples and continuous queries are originated at any node of the network. Nodes insert data in the form of relational tuples and each query q is represented in SQL. Tuples and queries are timestamped to represent the time that are inserted in the network by some node. Additionally, each query is associated with a unique key used to identify it in query grouping and to relate it to the node that submitted it.

2.2 Stream Processing Model

A data stream S_i is a sequence of tuples ordered by an increasing timestamp where $i \in [1..m]$ and $m \geq 2$ denotes the number of input streams. At each time unit, a number of tuples of average size l_i arrives to stream S_i. We use λ_i to denote the average arrival rate of a stream S_i in terms of tuples per second.

Many applications are interested in making decisions over recently observed tuples of the streams. This is why we maintain each tuple only for a limited time. This leads to a sliding window W_i over S_i that is defined as follows. Let $T(W_i)$ denotes the size of W_i in terms of seconds, i.e. the maximum time that a tuple is maintained in W_i. Let

TS(s) be a function that denotes the arrival time of a tuple $s \in S$, and t be current time. Then W_i is defined as $W_i = \{s | s \in S_i \wedge t\text{-}TS(s) \leq T(W_i)\}$.

Tuples continuously arrive at each instant and expire after $T(W_i)$ time steps (time units). Thus, the tuples under consideration change over time as new tuples get added and old tuples get deleted. In practice, when arrival rates are high and the memory dedicated to the sliding window is limited, it becomes full rapidly and many tuples must be dropped before they naturally expire. In this case, we need to decide whether to admit or discard the arriving tuples and if admitted, which of the existing tuples to discard. This kind of decision is made using a *load-shedding strategy* [14][21] which yields that only a fraction of the complete result will be produced.

2.3 Gossip Dissemination System

The basic idea behind gossiping [15][26] is to have all nodes collaborating to disseminate information into the entire network using a *partial view* stored in a *local list* of size *L*. To this end, when a node wishes to disseminate a message or receive it for the first time, it picks *k* nodes from its *local list* and sends them the message. Initializing and maintaining the local list at each node in face of dynamic changes in the network is done by a *membership protocol*. The number of gossip targets, *k*, is a typical configuration parameter called *fanout*. The sum of the *k* links between nodes specifies an overlay network on top of the existing network topology.

If a node receives a message for the second time, it simply discards the message. A relatively inexpensive optimization is to never forward a message back to the node it was just received from. Due to their inherent redundancy, gossip dissemination systems are able to mask network omissions and also node failures. In order to evaluate the performance of gossip dissemination systems, it is essential to define a set of metrics:

- **Hit Ratio**: the number of nodes that receive a sent message. Ideally, a gossip dissemination system should always achieve a hit ratio of 100%, that is, to reach every node in the network.
- **Dissemination Speed:** the time a message requires to reach every node in the network. It depends of network latency and the number of hops a message takes to reach the last node. In our evaluation we focus on the latter factor.
- **Message Overhead:** the number of redundant times that a message is forwarded during its dissemination.

An effective dissemination system assures a high Hit Ratio, a fast dissemination speed, and has a low message overhead.

2.4 Problem Definition

In this paper, we deal with the problem of processing join queries over data streams. A data stream is a sequence of tuples ordered by monotonically increasing timestamps. The timestamps are generated using the KTS service [1] which is a distributed service that deals with generating monotonically increasing timestamps in

DHTs. Each tuple and query have a timestamp that may either be implicit, i.e. generated by the system at arrival time, or explicit, i.e. inserted by the source at creation time. However, we do not include the timestamp attribute in the schema of the stream. Formally, the problem can be defined as follows. Let $\Sigma=\{S_1,S_2,\ldots..,S_m\}$ be a set of relational data streams, and $Q=\{Q_1,Q_2,....,Q_n\}$ be a set of join queries specified on these data streams. Our goal is to provide an efficient method to execute Q over Σ in terms of network traffic.

3 DHTJoin Method

Using structured overlay P2P networks in data stream processing environments is a novel research topic [13]. We can take advantage of it strengths to support queries over data streams [4]. The main issues for processing continuous queries in DHTs are the following: how to route data and queries to peers in an efficient way; how to provide a data storage mechanism for storing relational data; and how to provide a good approximate answer to join queries.

We describe our solution for continuous join query processing using DHTs. Our method has two steps: indexing of tuples and dissemination of queries. A tuple inserted by a node is indexed, i.e., stored at another node using DHT primitives only if there is a query that requires it. To assure the knowledge of that query to the entire network, it is disseminated using a gossip protocol.

3.1 Indexing Tuples

Existing DHT implementations such as Chord [22] are robust: they use simple algorithms with provable properties even under frequent node failures, departures and re-joins. To answer a query, we consider different kinds of peers. The first kind is Stream Reception Peers (SRP) which are responsible for disseminating queries and indexing tuples to the second kind of peers named Stream Query Peers (SQP). SQPs are responsible for executing queries over the arriving tuples using their local sliding windows, and sending the results to the third kind of node(s) named User Query Peer (UQP). To support dissemination of queries each node is a gossip node and for indexing of tuples each node is a DHT peer. Note that the distribution between SRP, SQP and UQP is functional and the same peer can have all of these functionalities.

The tuples arrive at any SRP and are indexed onto SQPs if there exists a query that requires it. A join processing algorithm is executed at each SQP that receives the tuples sent by an SRP. In our solution, the tuples are indexed in SQPs. Each arriving tuple is entirely indexed in an SQP using a join attribute value as storage key, usually in $O(log\ n)$ routing hops where n is the number of peers.

Let us describe our indexing method for three streams S_1, S_2 and S_3. However, our method is not limited to three streems and works on any number of streams. Let A be the set of attributes of S_1, S_2 and S_3. Let s_1, s_2, and s_3 be tuples belonging to S_1, S_2 and S_3 respectively and $val(s\ a)$ be a function that returns the value of an attribute $a \in A$ in tuple s. Let h be a uniform hash function that hashes its inputs into a DHT key, i.e. a number which can be mapped by the DHT onto a peer. For indexing a tuple s_1 that arrives at a SRP, each tuple obtains an index key computed as $key= h(S_1,val(s_1,a))$ and

is then indexed in a SQP using *put(key,s_1)* only if there exists a query that requires this tuple indexed by using the value of attribute *a*. For s_2 and s_3 tuples, we proceed in the same way. All tuples of S_1, S_2 and S_3 having the same value in attribute *a* get indexed at the same SQP. Join operations are performed on the tuples stored within sliding windows. Let W_i denote a sliding window on stream S_i. We use time-based sliding windows where $T(W_i)$ is the size of the window in time units. At time *t*, a tuple belongs to W_i if it has arrived in the time interval $[t- T(W_i),t]$.

To illustrate, let us consider an equijoin query over three streams S_1, S_2 and S_3. Once a S_1-tuple arrives onto a SRP we verify if a query requires this tuple indexed by some attribute, if so, we index the tuple and execute a Join processing algorithm at each SQP that receives SRP tuples. Processing a join query over timestamp based sliding windows with three input relational streams S_1, S_2 and S_3 is done as follows. Upon each arrival of a new tuple from stream S_1 to W_1, expired tuples in W_2 and W_3 are invalidated, then for each W_2-tuple we evaluate the join of W_3 with the arriving tuple and probe the result set. A load shedding procedure is executed over W_1's buffer if there is not enough memory to insert the tuple. The UQP receives the tuples from SQPs. At any time, a SQP could contain a collection of tuples representing a portion of the join query result. The join query result is inserted into a local queue and sent to the peer that submitted the query. The output of the join sent to that peer (an UQP node) consists of all pairs of tuples $s_1 \in S_1$, $s_2 \in S_2$ and $s_3 \in S_3$ such that $s_1.a= s_2.a=s_3.a$ and, a time *t*, both $s_1 \in W_1$, $s_2 \in W_2$ and $s_3 \in W_3$.

3.2 Disseminating Queries

A query *q* can originate at any of the nodes and is disseminated to the whole network. This dissemination allows a global knowledge of queries and to index tuples using an attribute value only if there exists a query that requires it. However, disseminating *q* can be difficult if nodes have only limited knowledge of the members of the network. This knowledge is stored in a *local list* of size *L* and we must assure a Hit Ratio of 100%. We assume a network of *N* nodes. Intuitively, the requirement for assuring a Hit Ratio of 100% is that the components of the *local list* form a strongly connected directed graph over all nodes. For example, if we use the Chord protocol we can build a *local list* of size 1 with an immediate successor pointer from the finger table and disseminate a query in *N* hops without redundant messages. However, the dissemination speed is very slow. To accelerate the dissemination process we can consider a *local list* composed by an immediate successor pointer from the finger table and the predecessor pointer, used to walk counter clockwise around the Chord ring, maintained by the Chord protocol. In this case, the dissemination speed is reduced to *N/2* hops with 2 redundant messages. Considering that a tuple can be indexed in *log(N)* hops we must increase the dissemination speed as high as possible. To this end, we can increase the *fanout* by exploiting the entire finger table and increase the dissemination speed to *log(N)* hops, i.e., we disseminate a query as fast as a tuple index but that implies redundant messages. Redundancy provides tolerance to dynamic environments but excessive redundancy leads to excessive network traffic.

In addition, with predecessor and successor pointers of the Chord protocol we can form a strongly bidirectional connected directed graph that assures a 100% Hit Ratio

and with the entire finger table, we can increase the dissemination speed as fast as a tuple index.

3.3 Optimization

Achieving a dissemination speed of queries equal to *log(N)* hops (taking advantage of the Chord finger table) yields an increase in the number of forwarded messages. We propose an optimization to decrease considerably the number of redundant messages without decreasing the dissemination speed and guaranteeing a 100% Hit Ratio. Our solution is as follows. Each time a node originates a query, it builds a gossip message that consists of the query, its own identifier and its gossip targets: the successor and predecessor pointers and its finger table. Upon reception of a new gossip message, a node builds its gossip targets considering the intersection of the nodes received with the arriving message, its own identifier and its own gossip targets. Thus, we can decrease the number of redundant messages to *1/2log(N)* (see our performance evaluation), while maintaining the same dissemination speed and assuring a 100% of Hit Ratio.

4 Performance Evaluation

To test our DHTJoin method, we implemented a simulator using SimJava [20] running on a Linux PC with a 3.4 Ghz Pentium Processor and 512 megabytes of RAM. Our simulator is based on Chord which is a simple and efficient DHT. To simulate a peer, we use a SimJava entity that performs all tasks that must be done by a peer in the DHT, in the dissemination system and in the stream window-based join processing.

We generate arbitrary input data streams consisting of synthetic asynchronous data items with no tuple-level semantics. We have a schema of 4 relations, each one with 10 attributes. In order to create a new tuple we choose a relation and assign values to all its attributes using a Zipf distribution with a default parameter of 0.9. The max value of the domain of the join attribute is fixed to 1000. Tuples on streams are generated at a constant rate of 30 tuples per second. Queries are generated with a mean arrival rate of 0.02, i.e., a query arrives every 50 seconds on average. The network size is set to 1024 nodes. In all experiments, we use time-based sliding windows of 50 seconds. For each of our tests, we run the simulator for 300 seconds. In order to assess our approach, we compare the network traffic of DHTJoin against a complete implementation of RJoin [13] which is the most relevant related work (see Section 5). RJoin uses incremental evaluation based on tuple indexing and query rewriting over distributed hash tables. In RJoin a new tuple is indexed twice for each attribute it has; wrt the attribute name and wrt the attribute value. A query is indexed waiting for matching tuples. Each arriving tuple that is a match causes the query to be rewritten and reindexed at a different node. The network traffic is the total number of messages needed to index tuples and disseminate a query in DHTJoin or to index tuples and perform query rewriting in RJoin.

In order to show the effectiveness of the dissemination of queries, we use the Hit Ratio, Dissemination Speed and Message Overhead metrics. The fanout parameter is

set to 2, *m*/2, *m* and m_{OPT}, with *m* equal to *log(N)*. The fanout value m_{OPT} includes the optimization process presented in Section 3.

In the rest of this section, we evaluate network traffic and the effectiveness of gossip dissemination.

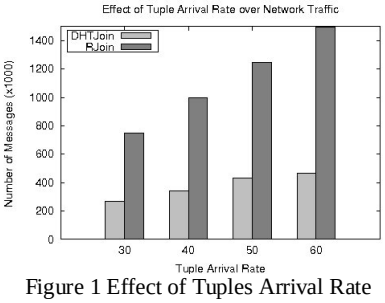

Figure 1 Effect of Tuples Arrival Rate

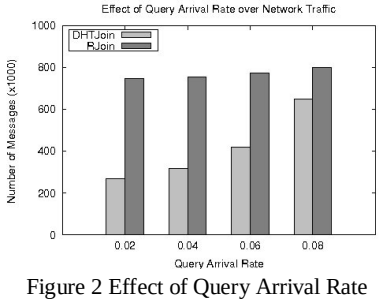

Figure 2 Effect of Query Arrival Rate

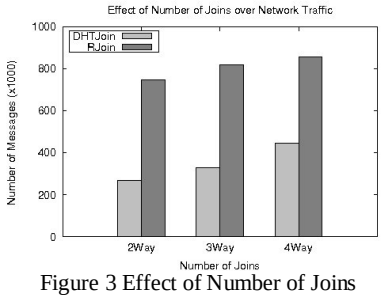

Figure 3 Effect of Number of Joins

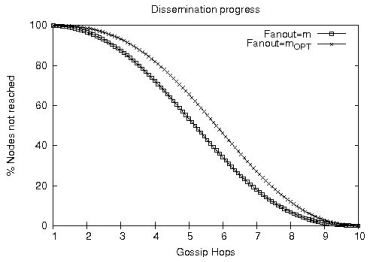

Figure 4 Dissemination progress

4.1 Network Traffic

In this section, we investigate the effect of tuples' arrival rate, query's arrival rate and number of joins on the network traffic. The network traffic of RJoin and DHTJoin grows as the tuples' arrival rate grows. In RJoin, as more tuples arrive, the number of messages related to the indexing of tuples and query rewriting increases (see Figure 1). As expected, DHTJoin generates significantly less messages because a high tuples' arrival rate does not mean more indexing of tuples. The reason is that before indexing a tuple, DHTJoin checks for the existence of a query that requires it. In Figure 2, we show that, as more queries arrive, RJoin generates more query rewriting messages. However, DHTJoin generates more messages only if new queries related to new different values used in the tuples arrive in the system. Figure 3 shows that more join require more network traffic. RJoin generates more query rewriting when there are more joins in the queries. However, in DHTJoin the network traffic increases only if the arriving queries require of attributes not present in the other queries. The reason is that with the dissemination of queries, DHTJoin can avoid the unnecessary indexing of tuples that are not required by the queries

In summary, due to the integration of query dissemination and hash-based placement of tuples our approach avoids the excessive traffic generated by RJoin which is due to its method of indexing tuples.

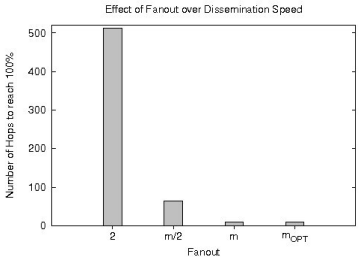

Figure 5 Speed of dissemination

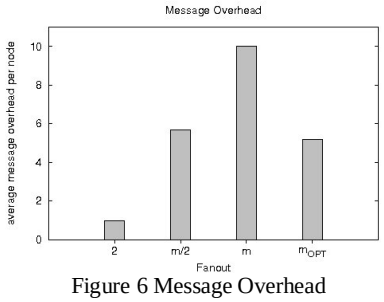

Figure 6 Message Overhead

4.2 Gossip Dissemination

To evaluate the effectiveness of gossip dissemination of queries, we post a query originated randomly at any node. In this experiment, we show the dissemination speed and look at the evolution of dissemination in terms of the number of nodes that have not yet been contacted as a function of the hops taken. Figure 4 shows that our method yieds a complete dissemination, i.e., a Hit Ratio of 100% in *log(N)* hops. This is because the nodes of the *local list* form a strongly connected directed graph over all nodes. Our proposed optimization to reduce message overhead produces no changes in the speed of the dissemination.

If a query is disseminated as fast as possible we avoid excessive store time of tuples waiting for a query and we can generate join tuples as early as possible. Thus, we state that a query must be disseminated as fast as a tuple. In Figure 5, we confirm this and we can see that the higher the fanout the dissemination speed increases to reach the same speed as a tuple index. However, if we choose a higher fanout, we incur message overhead which overload the network. Conversely, we can choose a lower fanout to obtain a minimal overhead, as shown in Figure 6, but this incurs slower dissemination, as shown in Figure 5. To reduce message overhead we ran experiments to include the optimization proposed in Section 3. In Figure 5, we show that with a fanout m_{OPT}, we obtain a fast dissemination and at the same time, as shown in Figure 6, we reduce the message overhead to *1/2log(N)* wrt a fanout *m*.

5 Related Work

A DHT can serve as the hash table that underlies many parallel hash-based join algorithms. However, our approach provides Internet-level scalability. Our work is related to many studies in the field of centralized and distributed continuous query processing [12][10][24][6][17]. In PIER [12], a query processor is used on top of a DHT to process one-time join queries. Recent work on PIER has been developed to process only continuous aggregation queries. PeerCQ [10] was developed to process continuous queries on top of a DHT, However, PeerCQ does not consider SQL queries and the data is not stored in the DHT. Borealis [24], TelegraphCQ [6] and DCAPE [17] have been developed to process continuous queries in a cluster setting and many of their techniques for load-shedding and load balancing are orthogonal to our work. The most relevant previous work regarding the utilization of a structured

overlay P2P network is [13] which proposes RJoin, an algorithm that uses incremental evaluation. This incremental evaluation is based on tuple indexing and query rewriting over distributed hash tables. A major difference in our work differs is that our tuple index mechanism indexes tuples only if there exists a query that requires it.

Large-scale information dissemination based on gossip protocols are essential for applications such as replicated database maintenance [8], publish/subscribe systems [9] and distributed failure detection [25]. These approaches do not consider a structured overlay setting**.** The probabilistic dissemination algorithm named Randcast proposed in [15] spreads messages very fast but fails to reach every node in the network. In [18] the authors propose a technique that leverages simple social network principles enabling nodes to select gossip targets intelligently. This work shows experimentally that gossip dissemination on the Chord overlay fails to deliver the messages to every node in the network. Our work instead shows that it is possible to reach every node in the network forming a strongly bidirectional connected directed graph composed by predecessor and successor pointers. In [16] the authors assure a good tradeoff between message overhead and reliability guarantee using a connection graph called a Harary graph. However, its principal drawback is the maintenance of such graph that requires global knowledge of membership. In our work, the structure that supports the membership protocol is supported by the structured overlay and does not require global knowledge of membership for its maintenance. The most relevant previous work regarding dissemination is the hybrid dissemination proposed in [26]. This works uses deterministic links to assure a complete dissemination. However, our work differs in the presence of a structured overlay and a membership protocol based on its routing table.

6. Conclusion

In this paper, we proposed a new method, called DHTJoin, for processing continuous join queries using DHTs. DHTJoin combines hash-based placement of tuples and dissemination of queries using a gossip style protocol. Our performance evaluation shows that DHTJoin obtains significant performance gains due to our schema of global knowledge of queries based on a gossip protocol. This schema has a low message overhead and avoids the excessive traffic produced by the tuple index method and the query rewriting of RJoin. Our results demonstrate that the total number of messages transmitted by DHTJoin is always fewer that RJoin wrt tuple arrival rate, query arrival rate and number of joins.

As future work, we plan to address the problem of efficient execution of top-k join queries over data streams using DHTs taking advantage of the best position algorithms [2] which can be used in many distributed and P2P systems for efficient processing of top-k queries.

References

1. R. Akbarinia, E. Pacitti and P. Valduriez. Data currency in replicated DHTs. *ACM Int. Conf. on Management of Data (SIGMOD)*, 211-222, 2007.
2. Akbarinia R., Pacitti E., Valduriez P. Best Position Algorithms for Top-k Queries. *Int. Conf. on Very Large Databases (VLDB)*, 495-506. 2007.

3. Arasu A., Babu S., Widom J. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Technical Report, DataBase Group, Stanford University, 2002.
4. Babcock B., Babu S., Datar M., Motwani R., Widom J. Models and Issues in Data Streams. *ACM Symp. on Principles of Database Systems (PODS)*, 2002.
5. Bonnet, P., Gehrke, J., and Seshadri, P. Towards Sensor Database Systems. *Int. Conf. on Mobile Data Management*, 2001.
6. Chandrasekaran S. et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *Conf. on Innovative Data Systems Research (CIDR)*, 2003.
7. Chen, J., DeWitt, D., Tian, F., Wang, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM Int. Conf. on Management of Data (SIGMOD)*, 2000.
8. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J. Epidemics Algorithms for Replicated DB Maintenance. *ACM Int. Conf. on Management of Data (SIGMOD)*, 1987.
9. Eugster, P., Guerraoui, R., Handurukande, S., Kermarrec, A., Kouznetsov, P. Lightweight Probabilistic Broadcast. *Int. Conf. Dependable Systems and Networks (DSN)*, 2001.
10. Gedik B., Liu L. PeerCQ: A Descentralized a Self-Configuring Peer-to-Peer Information Monitoring System. *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2003.
11. Golab L., Özsu T. Processing Sliding Windows Multi-Joins in Continuous Queries over Data Streams. *Int. Conf. on Very Large Data Bases (VLDB)*, 2003.
12. Huebsch R., Hellerstein J.M., Lanham N., Loo B.T., Shenker S., Stoica I. Queying the Internet with PIER. *Int. Conf. on Very Large Databases (VLDB)*, 2002.
13. Idreos, S., Liarou, E., Koubarakis, M. Continuous Multi-Way Joins over Distributed Hash Tables. *Int. Conf. on Extending Database Technology (EDBT)*, to appear, 2008.
14. Kang J., Naughton J.F. and Viglas S. Evaluating windows joins over unbounded streams. *IEEE Int. Conf. on Data Engineering (ICDE)*, 2003.
15. Kermarrec, A., Massoulié, L., Ganesh, A. Probabilistic Realiable Dissemination in Large-Scale Systems. *IEEE Trans. Par. Distr. Syst.*, 14(2), 248-258, 2003.
16. Lin, M,-J., Marzullo, K., Masini, S. Gossip versus Deterministic Flooding: Low Message Overhead and High-Reliability for Broadcasting on Small Networks. *Int. Symp. On Distributed Computing*, 2000.
17. Liu B., Jbantova M., Momberger B., Rundensteiner E.A. A dynamically Adaptive Distributed System for Processing Complex Continuous Queries. *Int. Conf. on Very Large Data Bases (VLDB)*, 2005.
18. Patel, Jay A., Gupta, I., Contractor, N. JetStream: Achieveing Predictable Gossip Dissemination by Leveraging Social Network Principles. *IEEE Int. Symp. on Network Computing and Applications (NCA)*, 2006.
19. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., and Shenker, S. A Scalable Content-Addressable Network. *Proc. of SIGCOMM*, 2001.
20. SimJava. http://www.dcs.ed.ac.uk/home/hase/simjava/
21. Srivastava U., Widom J. Memory-limited Execution of Windowed Stream Joins. *Int. Conf. on Very Large Data Bases (VLDB)*, 2004.
22. Stoica, I., Morris, R., Karger, D., Kaashoek, M., and Balakrishnan, H. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *Proc. of SIGCOMM*, 2001.
23. Sullivan, M., Heybey, A. Tribeca: A System for Managing Large Databases of Network Traffic. *USENIX Annual Technical Conf.*, 1998.
24. Tatbul N., Zdonik S. Window-Aware Load Shedding for Aggregations Queries over Data Streams. *Int. Conf. on Very Large Data Bases (VLDB)*, 2006.
25. Van Renesse, R., Minsky, Y., Hayden, M. A Gossip-Style Failure Detection Service. *ACM/IFIP/USENIX Int. Middleware Conf.*, 1998.
26. Voulgaris, S., Van Steen, M. Hybrid Dissemination: Adding Determinism to Probabilistic Multicasting in Large-Scale P2P Systems. *ACM/IFIP/USENIX Int. Middleware Conf.*, 2007.